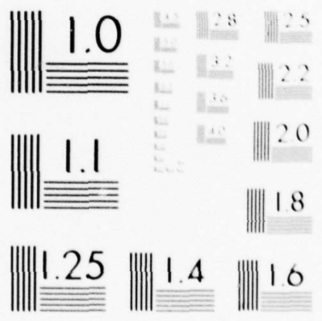
AD-A072 380 NORTHWESTERN UNIV EVANSTON IL DEPT OF ELECTRICAL ENG--ETC F/G 9/2
PERFORMANCE CONSIDERATIONS IN THE MAINTENANCE PHASE OF LARGE-SC--ETC(U)
JUN 79 S S YAU, J S COLLOFELLO F30602-76-C-0397

UNCLASSIFIED RADC-TR-79-129 NL

| OF |

AD
A072380

END
DATE
FILMED
9-79
DDC


1.0
2.8
2.5
3.2
2.2
3.6
4.0
2.0
1.1
1.8
1.25
1.4
1.6

LEVEL II

12

SC

RADC-TR-79-129
Interim Report
June 1979

PERFORMANCE CONSIDERATIONS IN THE MAINTENANCE PHASE OF LARGE-SCALE SOFTWARE SYSTEMS

Northwestern University

S. S. Yau
J. S. Collofello

ROME AIR DEVELOPMENT CENTER
Air Force Systems Command
Griffiss Air Force Base, New York 13441

RADC-TR-79-129 has been reviewed and is approved for publication.

APPROVED: Rocco F. Iuorno
ROCCO F. IUORNO
Project Engineer

APPROVED: [illegible]
WENDALL C. BAUMAN, Colonel, USAF
Chief, Information Sciences Division

FOR THE COMMANDER: John P. Huss
JOHN P. HUSS
Acting Chief, Plans Office

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

19 **REPORT DOCUMENTATION PAGE**		READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER 18 RADC-TR-79-129	2. GOVT ACCESSION NO.	3. RECIPIENT'S CATALOG NUMBER
4. TITLE (and Subtitle) 6 PERFORMANCE CONSIDERATIONS IN THE MAINTENANCE PHASE OF LARGE-SCALE SOFTWARE SYSTEMS		5. TYPE OF REPORT & PERIOD COVERED 9 Interim Technical Report, 1 Aug 76 - Nov 78
		6. PERFORMING ORG. REPORT NUMBER N/A
7. AUTHOR(s) 10 Stephen S. Yau James S. Collofello		8. CONTRACT OR GRANT NUMBER(s) 15 F30602-76-C-0397
9. PERFORMING ORGANIZATION NAME AND ADDRESS Northwestern University Dept. of Electrical Engineering & Computer Science Evanston IL 60201		10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS 62728F 55810278 17 12
11. CONTROLLING OFFICE NAME AND ADDRESS Rome Air Development Center (ISIS) Griffiss AFB NY 13441	16	12. REPORT DATE Jun[redacted]979
	11	13. NUMBER OF PAGES 44
14. MONITORING AGENCY NAME & ADDRESS (if different from Controlling Office) Same	12 52p.	15. SECURITY CLASS. (of this report) UNCLASSIFIED
		15a. DECLASSIFICATION DOWNGRADING SCHEDULE N/A

16. DISTRIBUTION STATEMENT (of this Report)

Approved for public release; distribution unlimited.

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

Same

18. SUPPLEMENTARY NOTES

RADC Project Engineer: Rocco F. Iuorno (ISIS)

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

Software maintenance, performance consideration, large-scale software systems, ripple effect analysis, performance changes, mechanisms of propagation, performance attributes, critical sections.

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

Maintenance of large-scale software systems is a complex and expensive process. Large-scale software systems often possess both a set of functional and performance requirements. Thus, it is important for maintenance personnel to consider the ramifications of a proposed modification from both a functional and a performance perspective.

In this report the possible effect of program modifications during the maintenance phase on the performance of large-scale software systems is analyzed.

over

DD FORM 1 JAN 73 1473

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

400 351

Mechanisms for the propagation of performance changes from one part of the system to another are identified, and the releationship among these mechanisms, performance attributes, critical program sections and performance requirements is also investigated. The development of a maintenance technique for predicting which performance requirements in the system may be affected by a proposed modification is outlined. This technique will enable maintenance personnel to incorporate performance considerations in their criteria for selecting the type and location of software modifications to be made, and to identify which performance requirements must be verified after the modification in order to insure that they have not been violated by the modification.

An additional report is planned for the formal description of the algorithms composing this maintenance technique.

CONTENTS

Page

LIST OF FIGURES

PERFORMANCE CONSIDERATIONS IN THE MAINTENANCE PHASE OF LARGE-SCALE SOFTWARE SYSTEMS

1. INTRODUCTION

The amount of the maintenance effort in the life cycle of large-scale software has been large and continuously increasing. It has been estimated that some 75% of data processing personnel are already taken up by maintenance [1]. Program maintenance includes error corrections, enhancements of capability, deletion of obsolete capabilities and changes in mission requirements [2,3,4]. Optimization is also a form of maintenance requiring the modification of code within individual modules, or possibly the structure of the complete system in order to improve its efficiency [5].

The maintainability of a system is a measure of the ease of making modifications to the system. In software, the effect of a modification may not be local to the location of the modification, but may also affect other portions of the system. There is a ripple effect from the location of the modification to the other parts of the system that are affected by the modification. One aspect of this ripple effect is logical or functional in nature. It involves the characterization of the system in terms of assumptions and decisions of program modules. When a change in a decision of a module is made, all of the modules in the system that have the modified decision as an assumption are affected. Modifications on these modules are usually necessary and these modifications may affect the decisions made in these modules. These modifications in the decisions will in turn affect other modules using them as assumptions. This ripple effect goes on until no decision is affected.

Another aspect of this ripple effect concerns the performance of the system. During software maintenance, it is possible to perform a modification to the system, investigate its logical ripple effect, and locate the other modules in the system affected by the modification. After all the logical corrections have been made to the system, the maintenance personnel may conclude that they have restored the system to its previous level of functional correctness. The performance of the system, however, may have been altered as a direct result of this maintenance activity. Since a large-scale program usually has both functional and performance requirements, the net result of the maintenance effort may be satisfactory to the functional requirements, but not to some performance requirements.

In many large-scale systems, the violation of a performance requirement is equivalent to a system error and, thus, requires further corrective action [6]. Consequently, it is important in the maintenance process to fully understand the potential effect of a modification to the system in terms of the performance of the parts of the system directly involved in the modification as well as those that are affected indirectly. The change in performance of these parts may then have an impact on the performance of other parts of the system. This ripple effect in terms of performance continues until the performance of no other part of the system is affected.

The maintenance process can, thus, be improved if maintenance personnel are supplied with information enabling them to incorporate performance considerations in their criteria for selecting the type and location of software modifications to be made. This information is provided by the development of a maintenance technique for predicting which performance requirements in the system may be affected by a proposed modification. The prediction of which performance requirements may be affected by a software modification is a difficult task. Due to the size and complexity of design of many large-scale systems, maintenance changes can cause repercussions almost anywhere throughout the system [7,8]. Thus, the significance of this technique lies in its ability to trace these repercussions and predict which performance requirements may be affected.

In this report, the development of a maintenance technique for predicting which performance requirements in the system may be affected by a proposed software modification will be outlined. Mechanisms for the propagation of performance changes, performance attributes, and critical software sections will be defined and their relationship with performance requirements will be analyzed. These results will form the basis for the development of the maintenance technique.

The technique outlined in this report is applicable to all types of large-scale systems possessing performance requirements including multiprocessing systems. The significance of the technique is its contribution to a software engineering approach to maintenance. It provides maintenance personnel with criteria for selecting the proper software modifications among the available alternatives. Since the performance requirements which may be affected by each proposed software modification can be identified using this technique, it is probable that the maintenance personnel will select modifications affecting the fewest number of performance requirements. If it is decided that some performance requirements are very close to being violated, the maintenance personnel may select the software modifications among available alternatives that do not affect these requirements in order to avoid further maintenance necessary to repair any violations to these performance requirements. Thus, in this report we will show how the maintenance technique is designed to complement the logical ripple-effect maintenance tool in the prediction of the repercussions generated by software modifications during the maintenance phase.

In this report we will also illustrate how the technique can help retest the system whether its performance requirements have been violated by the maintenance effort after the maintenance changes have been implemented. The maintenance technique analyzes the proposed changes in respect to the performance of the entire system, and not just the local areas involved in the modification. It can then determine the performance requirements affected by the maintenance change. It should be noted that during the evaluation of alternative modifications, the maintenance technique was used to predict worst-case effects on performance of software modifications. Now that the modification has been implemented, the maintenance technique can substantially refine its analysis and determine more accurately the performance requirements affected by the maintenance modifications. This permits the identification of which portions of the system must be retested to insure that these performance requirements have not been violated. Since the maintenance effort must not violate any functional or performance requirements, this maintenance technique provides a significant contribution in determining the scale of retesting effort needed to insure that these requirements have not been violated.

Throughout this report, we will identify the areas where more research is needed. In particular, we plan to have an additional report.

1. The second report will cover the following topics:

   a. Formal description of the algorithms for identifying the eight mechanisms for the propagation of performance changes in a large-scale program. Also included will be proofs of the correctness of these algorithms as well as illustrative examples.

   b. Formal description of the algorithms for identifying the critical software sections of a large-scale program.

   c. Formal description of the algorithms for identifying performance dependency relationships in a large-scale program.

   d. Formal description of the algorithms composing the technique for predicting performance requirements affected by maintenance activity. Also included will be proofs of the correctness of these algorithms.

   e. Demonstration of the maintenance technique during the maintenance phase of a typical program.

2. PERFORMANCE CONSIDERATIONS IN SOFTWARE LIFE CYCLE

Recent publications are beginning to stress the importance of performance considerations in the design phase of software development. Previous work in software specifications techniques has centered upon only the functional characteristics of software systems [9,10]. Performance considerations were only considered after the system was designed and implemented. Yet, failure to include performance specifications as part of the initial development process is one of the factors leading to significant problems during the development phase. This failure is a consequence of the "structured programming period" during which the prevalent attitude was that performance should only be dealt with after the system was built [11].

Performance analysis is now being introduced as an important and necessary tool for choosing which design among several alternatives should be used during the design stage. It is possible to conceive that more than one of the designs would satisfy the functional requirements of a problem, but the performance characteristics of the different designs will probably vary. In this case, it is the performance considerations that will determine which of the designs should be selected [12].

Performance considerations are also often in conflict with modularity considerations. In real-time systems, it is often critical to have the appropriate modules in memory when they are needed. This may conflict with the technique of designing a system to be functionally and logically modular [13]. Furthermore, performance requirements will sometimes impose additional constraints on the structure of the program so that principles in structured programming cannot be followed. For example, storage limitations and access time requirements may dictate that only certain data structures should be used. This will, however, violate Dijkstra's idea of postponing the choice of data structure to the last minute. Overall, program design and testing considerations often dictate modularity properties that often conflict with properties necessary for high performance [13]. This necessitates a tradeoff. The entire problem is summarized eloquently by Branscomb when he said, "The high cost of software has its roots in the lack of methodology for performance specification and evaluation" [14].

A current trend in software engineering is, thus, toward the inclusion of performance considerations in the evaluation of alternative designs in the design phase. Although current research results in this area are promising, there still remains a vast amount of work to be done [11,12,15,16]. Some of the preliminary results in this area that pertain to the study of performance considerations at the maintenance phase of the software life cycle will be noted as they are encountered. Performance considerations at the design phase, however, differ from those in the maintenance phase. They do share a common objective of identifying the best proposed alternative implementation in terms of performance. However, during the maintenance phase, the maintenance personnel must be able to ascertain the ramifications in terms of performance of proposed changes in order to achieve this objective.

3. LEVEL AT WHICH PERFORMANCE CHANGES DUE TO PROGRAM MAINTENANCE ARE CONSIDERED

In order to study the potential effect of a modification, it is first necessary to establish at what level of system decomposition the performance changes should be studied. Now the definition of what constitutes performance should be user-oriented because recognition of a change in performance of the system by a user is dependent upon his performance expectations. A minor change in performance may be unnoticed by most users, but the same change may be catastrophic to others. The ideal level of system decomposition for studying the potential effect of a program modification to performance then appears to be the level at which performance requirements can be stated and changes of performance meaningfully interpreted.

The approach to stating performance requirements has historically been in terms of functions and subfunctions as demonstrated in many military standards. This approach does have some difficulties when applied to large-scale software. First, it is often difficult to trace the processing required for an input message. Since functions may span subsystems and are often interrelated, it is also difficult to develop acceptance tests [17].

Another obvious way of stating performance requirements is in terms of the operations that each software module performs. The disadvantage of this approach is that the performance requirements specified in this way are typically oriented more towards the implementation techniques of the module rather than the user's requirements. In large-scale systems, some of the module performance requirements may be incompatible with others in the system. This can result in failure during the integration phase [18].

The current approach in software engineering to the statement of performance requirements is in terms of flows through the system [17-22]. The system is examined from the perspective of stimuli and responses. Each flow originates with a stimulus and continues to the ultimate response. Along each flow, performance requirements are stated to specify how well the functional requirements are met. The performance requirements are defined in terms of timing, response time, accuracy, frequency of occurrence, etc. for each stimulus-response pair. This enables simplified testing procedures for performance requirements stated in terms of specific paths and their associated data.

These performance requirements can be synthesized to form a Requirement Network (R-Net) composed exactly of the requirement paths [19]. An R-Net consists of:

(1) A set of nodes representing processing steps called "Alphas."

(2) A set of nodes identifying processing conditions, called "OR" nodes, associated with a selector variable.

(3) A set of "AND" nodes identifying "don't care" sequences of Alphas which could be performed in any sequence or even in parallel.

(4) A set of interface nodes where external messages are received or transmitted.

(5) A set of initial and terminal nodes.

(6) A set of arcs joining the nodes in a legal fashion.

(7) A set of events on R-Net paths, each of which enables an R-Net to perform processing. The arrival of a message at an interface may also constitute an event.

(8) A set of validation points used to uniquely identify paths, and to specify where data are to be extracted to test the process.

The R-Net is a synthesis of a set of paths. Each Alpha is associated with two subsets of data identifiers, representing data input to and output from the processing step. The link between the functional and performance requirements is achieved through the validation points. Each validation point is associated with a set of data identifiers representing data to be collected. If a test for a performance requirement is written only in terms of those data identifiers, then the requirement is testable [19].

A collection of R-Nets containing performance requirements for each path is called a Graph Model of Performance Requirements (GMPR) [19]. An example of a GMPR is shown in Figure 1. It should be noted that an R-Net does not necessarily describe paths through software modules. They also do not necessarily deal with the implementation of the software. Instead, they describe a sequence of required processes. Each of these processes may be implemented in a number of software modules, and some software modules may be implemented in several processes [18].

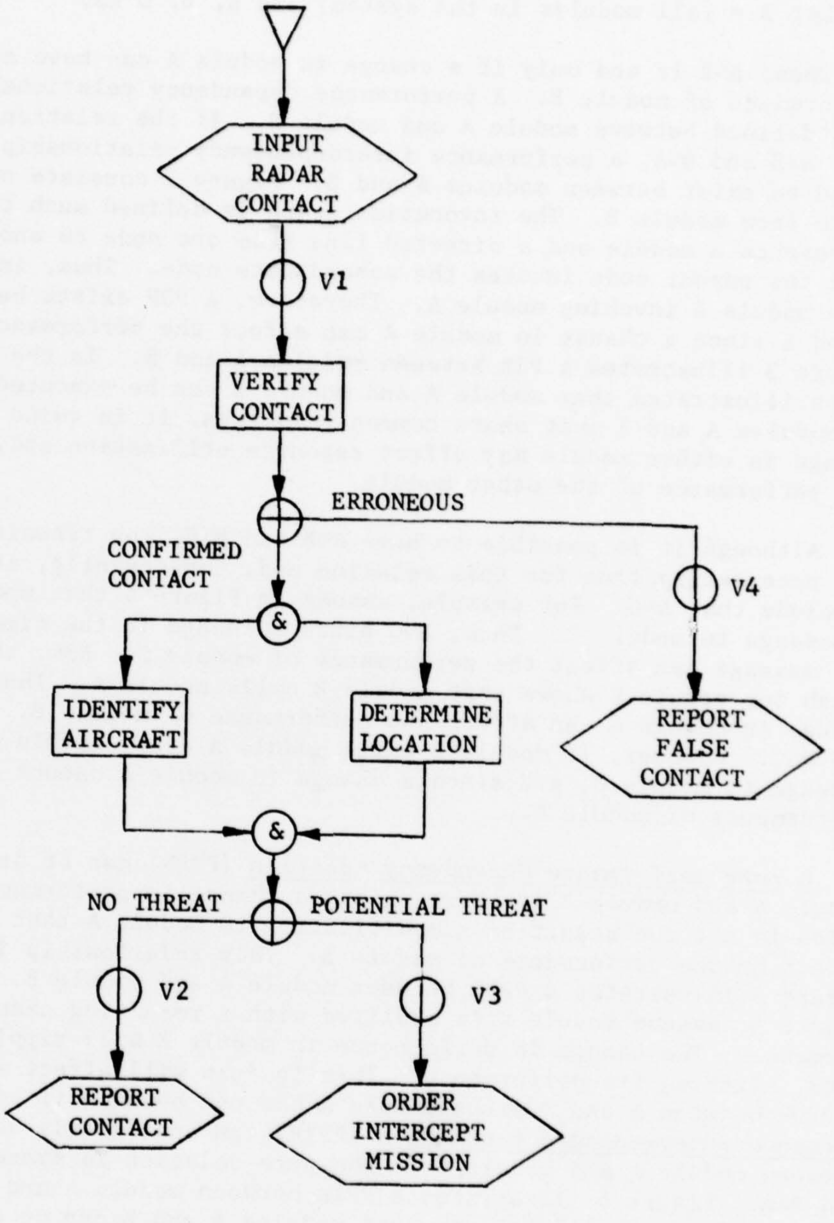
The development of the R-Nets associated with a system is a top-down process. The designer must first specify the responses in terms of a few processes stating the processing necessary at a high level. Later, each of these processes is in turn expanded into R-Nets. The refinement process continues until any further refinement would be machine or operating system dependent. The success of this approach to specifying the requirements of large-scale software has been demonstrated on systems as complex as the Ballistic Missile Defense systems [17,18].

Since the current trend in stating performance requirements is at the R-Net processing level, the potential effect of a software modification will be analyzed by the maintenance technique outlined in this report in terms of performance changes at this level. This level is also chosen because the recent development of requirement statement languages for expressing R-Nets provides a convenient interface with the maintenance technique [17,18,19]. Thus, the maintenance technique outlined in this report is designed to aid the maintenance effort of systems built with current design techniques as well as those systems which have been constructed utilizing previous design techniques. The maintenance technique is capable of handling both newly designed software and older existing software because it operates upon performance attributes which are decomposed from the performance requirements. When performance requirements are stated at the R-Net level, this decomposition is simpler than at other levels. As requirement statement languages continue to develop, it appears feasible that this decomposition may even be accomplished automatically. The performance requirements of existing software systems, which are not stated at the R-Net level can be decomposed into performance attributes, but this process may be more difficult to automate. Thus, although the maintenance technique is flexible enough to support the maintenance activity of all software systems, the ideal situation occurs when performance requirements are stated at the R-Net level.

A key objective in the maintenance phase is the identification of modules whose performance may change as a consequence of a software modification. These changes can then be placed in proper perspective by interpreting their effect on the performance requirements.

4. PERFORMANCE DEPENDENCY RELATIONS

The identification of modules whose performance may change as a consequence of software modifications is a complex task. The identification is complicated by the fact that performance dependencies often exist among modules which are otherwise functionally and logically independent. A relation "$\sim$" can be defined over the set of modules in the system as follows:



PATH P1 (V1-V4) Acknowledge false contact within 100 ms.

PATH P2 (V1-V3) Order intercept mission within 500 ms.

Figure 1. A graph model of performance requirements of a simple program expressed in terms of an R-Net.

Let S = {all modules in the system} and A, B, C εS,

Then, $A \sim B$ if and only if a change in module A can have an effect on the performance of module B. A performance dependency relationship (PDR) is then defined between module A and module B. If the relation is symmetric, i.e. $A \sim B$ and $B \sim A$, a performance interdependency relationship (PIR) is defined to exist between modules A and B. Figure 2 consists of an invocation graph from module B. The invocation graph is defined such that each node represents a module and a directed line from one node to another indicates that the parent node invokes the subordinate node. Thus, in Figure 2, we have module B invoking module A. Therefore, a PDR exists between modules A and B since a change in module A can affect the performance of module B. Figure 3 illustrates a PIR between modules A and B. In the figure, the graph illustrates that module A and module B can be executed in parallel. If modules A and B must share common resources, it is quite likely that a change in either module may affect resource utilization and, consequently, the performance of the other module.

Although it is possible to have $A \sim B$ and $B \sim C$, the transitive property is not necessarily true for this relation and, consequently, it is false to conclude that $A \sim C$. For example, assume in Figure 4 that module B is sending a message to module C. Thus, $B \sim C$ since a change in the time module B sends the message can affect the performance of module C. Now, the invocation graph for module B shows that module B calls module A. Thus, $A \sim B$ since a change in module A can affect the performance of module B. We now have $A \sim B$ and $B \sim C$. However, if module B calls module A after module B transmits the message to module C, $A \nsim C$ since a change in module A cannot affect the performance of module C.

A <u>pure</u> <u>performance</u> <u>dependency</u> <u>relation</u> (PPDR) can be defined between module A and module B if there exists a change in performance of module A which is not the result of a modification to module A that can have an effect on the performance of module B. This relationship is denoted by $A \approx B$. Figure 5 illustrates a PPDR between module A and module B. In the example in Figure 5, assume module X is modified with a resulting change in its performance. The change in performance in module X will ripple to module A, thus, altering its performance, This in turn will affect module B indicating a PPDR between A and B since module A has not been modified. A <u>pure</u> <u>performance</u> <u>interdependency</u> <u>relation</u> (PPIR) can analogously be defined to exist between module A and module B if the pure relation is symmetric, i.e., $A \approx B$ and $B \approx A$. Figure 6 illustrates a PPIR between module A and module B. In the figure, the graph illustrates that modules A and B can be executed in parallel. If modules A and B must share some common resources, it is quite likely that a performance change in either module may affect resource utilization and, consequently, the performance of the other module. If called module X within module A utilizes a data abstraction, a change in the implementation of the abstraction may affect the performance of X. This in turn may affect the performance of A. Thus, $A \approx B$. In an identical manner it can be shown that $B \approx A$. Therefore, a PPIR exists between module A and module B.

Two modules are defined to be <u>performance</u> <u>independent</u> (PI) if there does

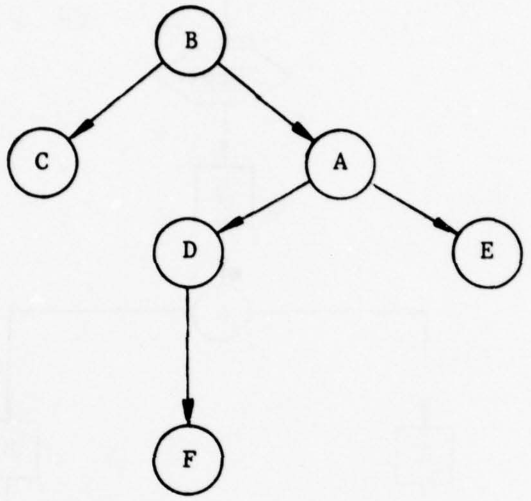
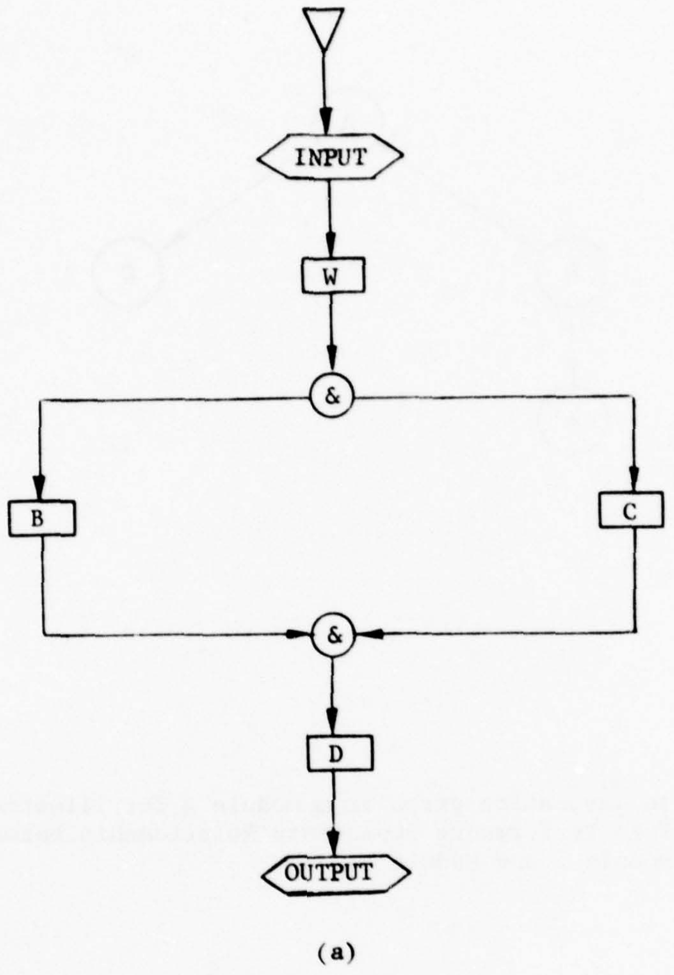


Figure 2. An invocation graph from Module B for illustrating a Performance Dependency Relationship.

Figure 3. An R-net of a simple program for illustrating a Performance Interdependency Relationship between Modules A and B.



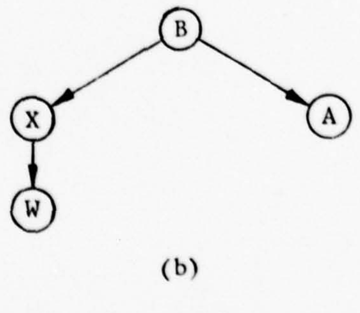


Figure 4. (a) The R-Net of a simple program and (b) the invocation graph from Module B for illustrating that the Performance Dependency Relationships are not transitive.

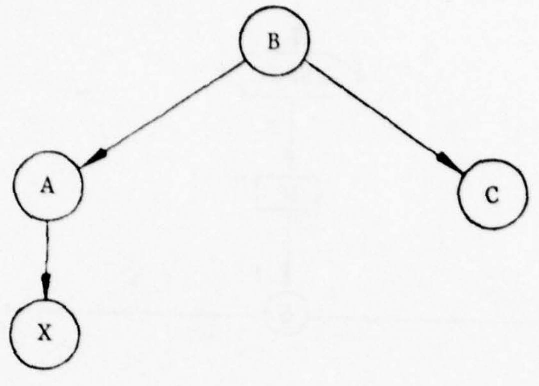


Figure 5. An invocation graph from module B for illustrating a Pure Performance Dependency Relationship between Module A and Module B.

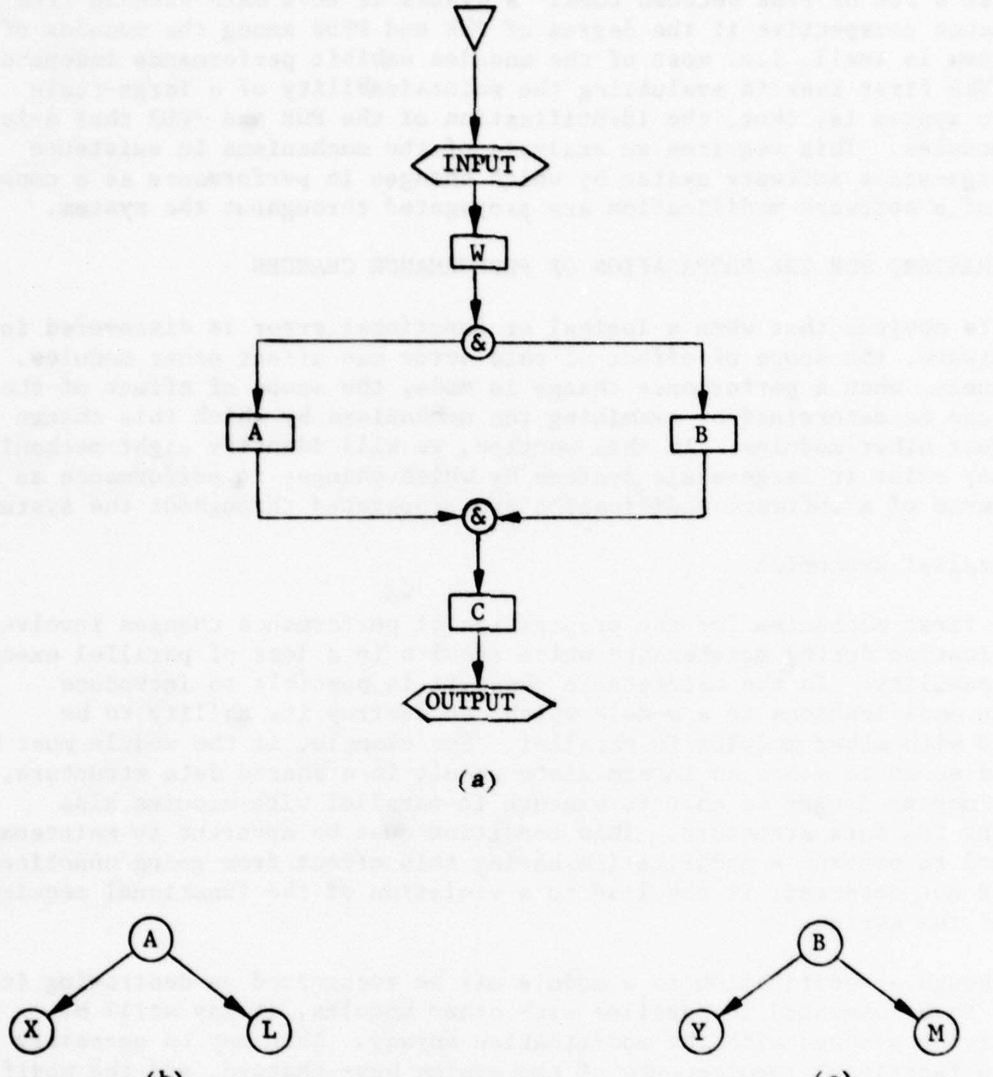


Figure 6. (a) The R-Net of a simple program, (b) the invocation graph from Module A, and (c) the invocation graph from Module B for illustrating a Pure Performance Interdependency Relationship between modules A and B.

not exist a PDR or PPDR between them. A system is more maintainable from a performance perspective if the degree of PDR and PPDR among the modules of the system is small, i.e. most of the modules exhibit performance independency. The first task in evaluating the maintainability of a large-scale software system is, thus, the identification of the PDR and PPDR that exist among modules. This requires an analysis of the mechanisms in existence in a large-scale software system by which changes in performance as a consequence of a software modification are propagated throughout the system.

5. MECHANISMS FOR THE PROPAGATION OF PERFORMANCE CHANGES

It is obvious that when a logical or functional error is discovered in the software, the scope of effect of this error can affect other modules. Analogously, when a performance change is made, the scope of effect of the change can be determined by examining the mechanisms by which this change can affect other modules. In this section, we will identify eight mechanisms which may exist in large-scale systems by which changes in performance as a consequence of a software modification are propagated throughout the system.

5.1 Parallel Execution

The first mechanism for the propagation of performance changes involves a modification during maintenance which results in a loss of parallel execution capability. In the maintenance phase it is possible to introduce software modifications to a module which can destroy its ability to be executed with other modules in parallel. For example, if the module must be modified so as to store an intermediate result in a shared data structure, then it may no longer be able to execute in parallel with modules also utilizing the data structure. This condition must be apparent to maintenance personnel to prevent a modification having this effect from going unnoticed. If it is not detected, it can lead to a violation of the functional requirements of the system.

Although a modification to a module may be recognized as destroying its ability to be executed in parallel with other modules, it may still be necessary to proceed with the modification anyway. This may be necessary when the functional requirements of the system have changed, and the modification is required to satisfy the new functional requirements. The maintenance personnel must then be aware of the change in performance which will result from the loss of the parallelism. Major changes in performance may result due to execution delays and contention for resources previously alleviated through the parallel execution. Thus, modifications affecting parallel execution of modules can lead to violations of performance requirements.

5.2 Shared Resources

Another mechanism for the propagation of performance changes is contention for resources among modules. When modules are forced to share resources, the time when each module requests and releases common resources are important performance parameters. In a multiprogramming environment, performance degradation may be experienced by modules whose execution is being affected by the denial of requested resources which are currently dedicated to other modules. The problem is intensified when the modules can execute in parallel. For example, consider the R-Net of a simple program in Figure 7. In the graph, assume that modules B and C must share a common resource. Then, a modification to the performance of either could affect the utilization of the common resource and, thus, the performance of the other. This illustrates a PIR between modules B and C since a change in module B can affect the performance of module C and a change in module C can affect the performance of module B. Thus, software modifications producing performance changes in the time resources are utilized can have detrimental effects on the performance of modules that must also share the resources.

5.3 Interprocess Communication

Another mechanism for the propagation of performance changes involves communication among the modules in the system.

When one module must send a message to another module, the performance of the module receiving the message is dependent upon when the message is actually received. Thus, modifications to the module sending the message that alter the time when the message is sent can affect the performance of the module designated to receive the message. This is another example of a PDR between the communicating modules. Since the key dynamic attribute in this environment is the time when the message is sent, it is possible for the communicating modules to participate in a PPDR. This situation is illustrated in the example in Figure 8. In the example, suppose module B must wait for a message from module A. If module A utilizes a data abstraction, and the implementation of that abstraction is modified, the time module A sends the message to module B may be affected. This in turn will affect the performance of module B. This implies a PPDR between module A and module B.

5.4 Called Modules

Another mechanism for the propagation of performance changes is the utilization of called modules in the software. Modifications to modules in the maintenance phase can be divided into two types. A bounded modification to a module is a modification which does not alter the performance of the module. An unbounded modification to a module is a modification which alters the performance of the module. An unbounded modification to the called module will affect the performance of all modules calling it. A bounded modification will not affect the performance of the other modules calling it.

5.5 Shared Data Structures

Another mechanism for the propagation of performance changes is through the utilization of shared data structures. In this mechanism, it is assumed that the implementation of any data abstractions utilized is not modified. Instead, changes in the contents of the data structures are analyzed in order to understand how they can affect the performance of other modules. The modules under investigation will be those utilizing the shared data structure. The basic dynamic attributes contributing to performance in this area are a

Figure 7. The R-net of a simple program for illustrating shared resources mechanism.

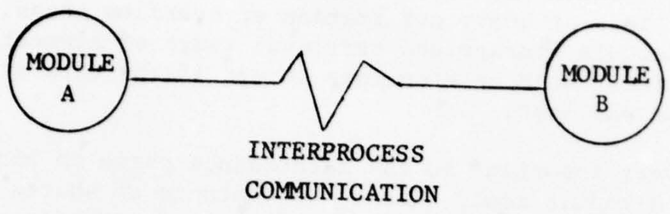


Figure 8. An example of the interprocess communication mechanism between two modules.

module's storage and retrieval times for entries in the data structures. Factors affecting storage and retrieval times vary among the different types of data structures as well as the algorithms utilized to manipulate these structures. For example, in linked structures the length of the lists can affect storage and retrieval times. In general, data structures and the algorithms to manipulate them are designed for a particular operational environment. During software maintenance, however, this environment may be altered. The modification may lead to saturation of the data structure resulting in long lists or heavy utilization of overflow areas. This may in turn reflect upon module storage and retrieval times of elements to the data structure. The effect could be even more severe if the data structure overflowed and data was lost.

It is, thus, very important in the maintenance phase to understand how a modification of a module could affect the contents of shared data structures. Modifications which affect the quantity of intermediate data stored must be analyzed as well as changes to modules which process and ultimately delete entries from data structures. For example, consider a queue shared between module A and module B. Assume module A generates data and stores it in the queue, and that module B processes this data and deletes it from the queue. Then either a modification to module A affecting the rate of intermediate data generated, or a modification of the processing of the data in module B can lead to overflow of the queue. The problem is even more acute when the data is perishable, i.e. it must be processed within a certain period of time. This problem is vividly exemplified in Air Force command and control systems. Due to the speed with which weapons can inflict destruction, decision times in command and control systems are very short. Also, much of the data required for dynamic force management is highly perishable and must be properly sequenced with other time-sensitive data to permit an accurate and useful picture of the situation [23]. Thus, modifications to modules sharing common data structures can lead to both performance and functional requirements violations.

5.6 Sensitivity to the Rate of Input

Another mechanism for the propagation of performance changes involves the rate of input into a process. This change in input frequency is the result of a changing environment. The resulting change of input rate to the process can have major repercussions in terms of its functional and performance requirements. For example, it can lead to saturation and possibly overflow of data structures involved with the processing of the input. The increased frequency of input arrivals may also lead to interruptions in processing which can lead to both functional and performance requirement violations.

These are difficult problems for maintenance personnel to address. When the software processing environment is modified resulting in increased rates of input, the maintenance personnel must evaluate this change in terms of its impact on the system processes which must handle the input. Thus, changes in the rate of input to processes are important and their effect on software functional and performance requirements must be evaluated.

5.7 Execution Priorities

Another mechanism for the propagation of performance changes involves the execution priority of modules. During the development phase, module execution priorities may have been established to insure correct sequencing or preservation of critical system performance requirements. The priorities are used to determine the execution order of modules capable of beginning execution at the same time. The priorities may also be utilized in the determination of whether or not a module's execution should be pre-empted for that of a module with a higher priority. During the maintenance phase, it is important for maintenance personnel to recognize the effect of a proposed modification in respect to the existing priorities in the system. For example, if module A has the ability to interrupt the execution of module B, then any modification affecting the execution time of module B must be carefully analyzed in order to determine if module B can still perform its designated function before being interrupted. Maintenance changes involving resetting of priorities or the addition of new priorities are even more difficult to assess. Modification of existing priorities can create conflicts in the system such as resource contention that can lead to performance degradation. Thus, priority considerations are important in the preservation of the performance requirements of the system.

5.8 Abstractions

Another mechanism for the propagation of performance changes is the utilization of abstractions in the software. The use of abstractions is a popular design tool and adds to the maintainability of the system by hiding design decisions. From the performance perspective of maintainability, however, abstractions are "trojan horses." This is because a change in the implementation of the abstraction will very likely affect the performance of the abstraction, and, thus, the performance of all modules utilizing the abstraction. For example, a data abstraction may have an associated retrieval time for finding a particular entry. If the data structure is reorganized and different algorithms are implemented, the associated retrieval time may vary. In a module that is heavily utilizing the abstraction, the change in performance may be intolerable. This is a classic example of a case where a modification to software during maintenance does not produce any functional or logical changes, but it does result in performance changes.

6. IDENTIFICATION OF PERFORMANCE CHANGE PROPAGATION MECHANISMS IN SOFTWARE SYSTEMS

The next task that must be completed is to find whether the eight mechanisms discussed in the last section may exist in a software system under study. This involves the development of algorithms for the determination of which mechanisms are in effect and which modules are within their influence. With this information available, it will be possible to determine what performance dependency relationships exist in the software system. This is the information which is vitally needed by maintenance personnel in the evaluation of the impact of proposed software changes on performance requirements.

The identification of most of the performance propagation mechanisms in a program can begin as early as the design stage. At this early stage, the major input into the algorithms to identify the existence of these mechanisms in a program consists of a set of R-Nets and invocation graphs for the program as well as some designer supplied information such as execution priorities. It is then possible to perform an analysis of the program to determine which mechanisms are present and, consequently, where performance dependencies exist. A complete static analysis of the implemented program is necessary for a more precise prediction of performance changes resulting from software modification.

In this section, algorithms for the determination of which mechanisms are in effect in a large-scale program will be briefly discussed. In another report, we plan to formally describe each of these algorithms with accompanying examples and proofs of the correctness of the algorithms.

6.1 Identification of Modules Executable in Parallel

The first mechanism for the propagation of performance changes to be examined in a software system is a change involving a loss of parallel execution capability. During the maintenance process, programmers must constantly be aware of which modules can be executed in parallel. They must then be cautious about modifications to these modules that can destroy their ability to execute in parallel. If the ability to execute in parallel must be lost in order to incorporate some type of modification, then it is important to determine the performance changes that will result in other modules in the system. The major performance change will be experienced in the process in which the module was executed in parallel. The primary effect will be an increase in execution time as a consequence of the lost parallelism. The delay can be considerable if the modified module must wait for resources that were previously available at its earlier execution time. Thus, information regarding changes in performance as a consequence of lost parallelism must be available to maintenance personnel in order that they can fully ascertain the implications of proposed modifications. This information is dependent upon the identification of which modules may be executed in parallel. The determination of which modules can be executed in parallel is a decision made during the design phase of the system. This decision must be reflected in either the software implementation or its accompanying documentation. In either case, it can be illustrated through the use of R-Nets. Since the identification of which modules may execute in parallel is an important step in many of the algorithms, a common algorithm will be briefly discussed for accomplishing this objective.

The identification of which modules may be executed in parallel based upon the information in an R-Net is easy in a small program. For example, in the program illustrated in Figure 9, it is obvious that modules B and C, C and D, and H and I are the only modules which may be executed in parallel. A convenient notation for describing which modules may be executed in parallel is to enclose those modules that may be executed in parallel within parentheses. For example, (A,B) denotes that modules A and B may be executed in parallel. The same notation can be modified when only a single module in a set may be executed in parallel with a single module in another set.

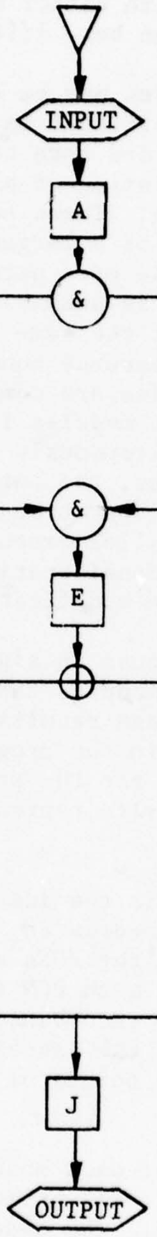


Figure 9. The R-net of a simple program for illustrating the identification of modules executable in parallel.

The individual modules are then replaced by the sets of which they belong in the notation. For example, ({A,B},{C,D}) represents that either module A or B may be executed in parallel with either module C or D. The notation for the example in Figure 9 would then be: {({B,D}C), (H,I)}.

The identification of which modules may be executed in parallel based upon the R-Net graphs becomes more difficult as the complexity and size of the program increases. In larger and more complex programs, it is also important to determine the sets consisting of all modules that can be executed in parallel at the same time. These sets can be defined as parallel execution sets. An example of a larger and more complex system is illustrated in Figure 10. One example of a parallel execution set from this example is {E,F,C,I,J} since it is possible for each of the modules in the set to be executed in parallel at the same time. Parallel execution sets play an important role in the resource contention problem, since it is necessary to determine how many modules are competing for the fixed number of available resources. Although the modules in parallel execution sets may seldom actually be executed simultaneously in the system due to their current relative execution start times, the potential for their simultaneous execution exists. After a period of operation and maintenance, it is possible that the modules in the parallel execution sets could be executed simultaneously and, therefore, this consideration should remain a factor in determining the effect of software modifications.

We would like now to briefly discuss an algorithm to identify for each module of a program the set of modules that may be executed with it in parallel in the program. From these results, it will then be possible to find the parallel execution sets in the program. The major input to the algorithm consists of an R-Net graph for the program. The graph should be at the level of abstraction necessary to represent the program at the module level.

The first step of the algorithm is the identification of parallel control nodes (PCNs). A PCN is defined as an "AND" node in the R-Net with an out-degree greater than one, i.e. the PCNs signal the beginning of parallel execution. Associated with each PCN in the graph, there is at least one "AND" node for synchronizing the recombination of control flows eminating from the PCN. The identification of the associated "AND" nodes for each PCN is accomplished by examining the points of intersection of the control flows eminating from the PCN.

The next task to be performed for each module is the identification of which control flows pass through it. All predecessor PCN nodes on these control flows can then be identified. The associated "AND" nodes can then be identified. Those PCNs with "AND" nodes which are predecessors of the module under investigation are eliminated. The set of modules that can be executed in parallel with the module under consideration is then formed by adding to the set all modules on control flows eminating from the remaining PCNs and ending with the corresponding "AND" nodes on the control flow of the module under consideration.

After each module in the system has had its set of modules that can be

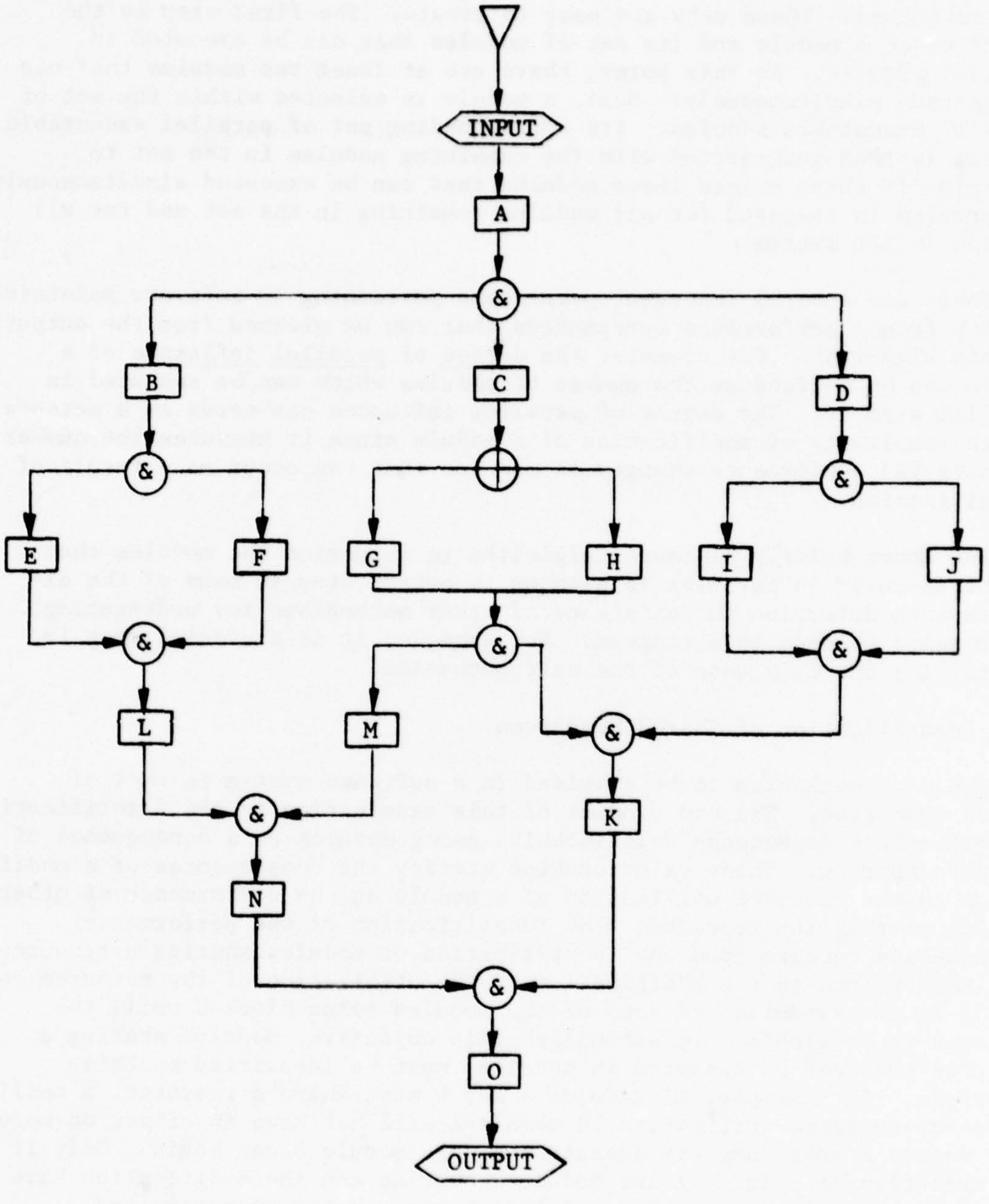


Figure 10. R-net of a larger and more complex program.

executed in parallel with it identified, the parallel execution sets can then be formed. These sets are easy to create. The first step is the selection of a module and its set of modules that can be executed in parallel with it. At this point, there are at least two modules that can be executed simultaneously. Next, a module is selected within the set of parallel executable modules. Its corresponding set of parallel executable modules is then intersected with the remaining modules in the set to determine if there exists three modules that can be executed simultaneously. The process is iterated for all modules remaining in the set and for all modules in the system.

There are several important parameters pertaining to software maintainability from a performance perspective that can be gleaned from the output of this algorithm. For example, the degree of parallel influence of a module can be defined as the number of modules which can be executed in parallel with it. The degree of parallel influence can serve as a measure of the complexity of modification of a module since it measures the number of potential performance changes in modules that can occur as a result of a modification.

The above briefly discussed algorithm to determine the modules that can be executed in parallel is also an important step in many of the algorithms to determine the existence of other mechanisms for propagating performance changes in a program. For example, it is a crucial step in identifying the existence of the next mechanism.

6.2 Identification of Shared Resources

The next mechanism to be examined in a software system is that of shared resources. The end product of this examination is the identification of performance dependency relationships among modules as a consequence of resource sharing. These relationships specify the consequences of a modification to the resource utilization of a module on the performance of other modules sharing the resource. The identification of the performance dependencies centers upon the identification of modules sharing a resource in a manner such that a modification of the utilization of the resource can result in the execution of some of the modules being blocked until the resource is available. To accomplish this objective, modules sharing a resource that can be executed in parallel must be identified by this algorithm. For example, if modules A and B must share a resource, a modification of resource utilization in module A will not have an effect on module B if module A must complete execution before module B can begin. Only if the execution of module A and module B overlap can the modification have an effect on performance. The modules sharing common resources and executable in parallel are then identified as being part of a performance interdependency relationship since a modification to one of the modules affecting its resource utilization can affect the performance of the other modules.

6.3 Identification of Interprocess Communication

The next mechanism for the propagation of performance changes to be identified in the software system involves interprocess communication. Interprocess communication is usually established in the design phase of the software system. These decisions involving interprocess communication must be reflected in either the software implementation or its accompanying documentation. Petri nets are a form of documentation that may be utilized in describing communications in the system [24]. Interprocess communication can also be recognized in the software when synchronization primitives such as P and V operators or WAIT and POST macros are utilized. It is then possible to perform a static analysis of the system to identify the modules involved in the communication. A performance dependency relationship can then be established between the modules sending the message and the modules receiving the message. This information can be saved for later utilization by maintenance personnel in their determination of the performance ramifications of proposed software modifications.

6.4 Identification of Called Modules

The next mechanism in a program for the propagation of performance changes to be studied is that of called modules. It is very easy to identify those modules in the system that are called by other modules by performing a static analysis of the system. Called modules can then be identified and performance dependency relations established between the called modules and the modules which call them. This information is important for maintenance personnel considering modifications to called modules in a program.

6.5 Identification of Shared Data Structures

The next mechanism in a program for the propagation of performance changes to be considered is that of shared data structures. A straight forward method of identifying this mechanism in a program would be to find all shared data structures and the corresponding modules manipulating them. This estimation, however, is not very accurate since only modules manipulating the data structure in a manner that may affect the performance of other modules utilizing this data structure are being investigated. As previously discussed, the basic attributes contributing to performance in this area are a module's storage and retrieval times for entries in the data structure. The factors affecting storage and retrieval times vary among the different types of data structures as well as the algorithms utilized to manipulate these structures. For example, a factor that is significant in many types of data structures is the number of entries stored in the structure.

The modules manipulating shared data structures can be classified into four categories based upon their utilization of the data structure. The categories are:

1. Reference entries only

2. Update entries

3. Create new entries

4. Delete old entries

It is, of course, possible for a single module to exist in more than one category.

The algorithm for identifying performance dependencies in this area is based upon the general notion that the number of entries in the data structure affects storage and retrieval times. One step of the algorithm would then be the classification of the modules sharing the data structure according to the above four categories. Performance dependency relationships could then be established between the modules creating the deleting entries and the other modules sharing the data structure. The relationships are valid since a modification to the modules creating or deleting entries may result in an increase in the number of entries in the data structure. This may affect storage and retrieval times, and ultimately the performance of the other modules sharing the data structure.

6.6 Identification of Modules Sensitive to Their Rate of Input

The next mechanism in a program for the propagation of performance changes to be studied is that of changes in the rate of input to a process. One of the major factors that determine whether a module is sensitive to its rate of input involves the module's storage structure for storing the input. Modules with fixed size storage structures may experience overflow problems. An increase in the rate of input may also result in the loss of previous input values. Another problem may be that the module has insufficient time to process the input before being interrupted by another input. An increase in the rate of arrival of inputs may also produce delays in servicing these inputs. Thus, the potential for performance requirement violations is large when the rate of input to a process is increased.

6.7 Identification of Execution-Priority Sensitive Modules

The next mechanism for the propagation of performance changes to be identified in the software involves execution priorities. As previously discussed, execution priorities are set during the software development phases to insure correct sequencing or preservation of critical system performance requirements. These priorities must be reflected in either the software implementation, in particular the dispatching algorithms, or in the accompanying documentation. The relative priority of modules executable at the same time can then be compared. A performance dependency relationship can then be considered in existence between a higher priority module and another which can execute at the same time. This relationship is important to maintenance personnel in determining the effect of a proposed modification on the performance of other modules in light of the existing execution priorities.

6.8 Identification of Abstractions

The next mechanism for the propagation of performance changes to be identified in a program is that of the "trojan horses," i.e. the abstractions. The utilization of abstractions in a module can be easily identified by static analysis. Abstractions can be recognized in the module as subroutine calls, function calls, and macros. Performance dependency relationships can then be established between the implementations of the abstractions and the modules utilizing them. This information will be valuable in aiding maintenance personnel evaluate possible changes of performance as a consequence of a modification of an abstraction implementation.

7. PERFORMANCE ATTRIBUTES

Performance attributes of a program are defined as attributes corresponding to measurements of key portions of the execution of the program. For example, one performance attribute of a module is its execution time. Another is the utilization for a particular resource during the execution of the program. There is a distinct relationship between performance attributes and the eight mechanisms for the propagation of performance changes. The eight mechanisms operate as links between performance attributes of modules. In other words, a change in a performance attribute of one module can affect a performance attribute in another module via one of the eight mechanisms. For example, let X represent the performance attribute corresponding to the time a resource is seized by module A. Assume module B is in contention for the same resource with module A and let Y represent the performance attribute corresponding to the time module B seizes the same resource. Then a change in performance attribute X can affect performance attribute Y via the shared resources mechanism.

The relationship between performance attributes and the eight mechanisms for the propagation of performance changes is illustrated in Figure 11, where the directed line labeled with a mechanism connecting two performance attributes indicates a performance dependency relationship exists between the performance attributes. For example, if performance attribute two of module A is modified, it can affect performance attribute two of module B via mechanism one.

We will now present twelve software performance attributes. These performance attributes are not a complete set of attributes corresponding to measurements of the execution of the program. Instead, these performance attributes are the attributes linked with the eight mechanisms as previously discussed. For a given module, not all of these performance attributes may be applicable.

Performance Attribute 1: The ability of the module to execute in parallel with another module.

Performance Attribute 2: For each resource in contention, the relative time that the module seizes the resource.

Performance Attribute 3: For each resource in contention, the relative time that the module releases the resource.

Performance Attribute 4: The relative time that the module begins execution.

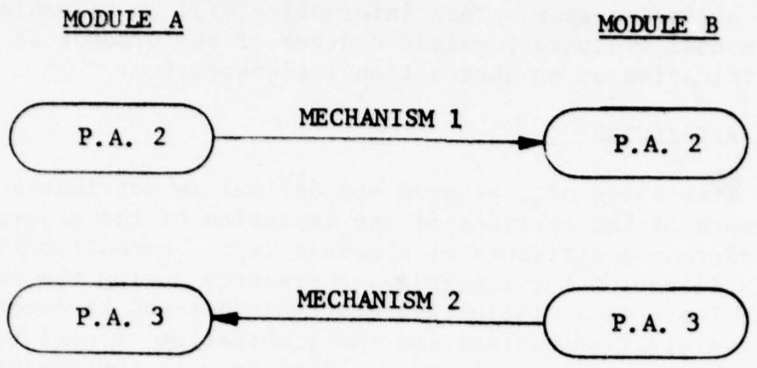


Figure 11. **Relationship of Performance Attributes and the Mechanisms for the Propagation of Performance Changes.**

Performance Attribute 5: The relative time that the module transmits a message to another module.

Performance Attribute 6: The execution time of the module.

Performance Attribute 7: For each resource utilized in the module, the resource utilization by the module.

Performance Attribute 8: For each dependent iterative structure in the module, the number of iterations.

Performance Attribute 9: For each data structure, the storage and retrieval times for entries in the data structures.

Performance Attribute 10: For each data structure, the number of entries in the data structure.

Performance Attribute 11: For each data structure, the service time of an entry in the data structure, i.e. the relative time that an entry remains in the data structure before being serviced.

Performance Attribute 12: The rate of input to the module.

8. CRITICAL SECTIONS OF A PROGRAM

Since the performance attributes of a program correspond to measurements of key portions of the execution of the program, they can be affected during the maintenance process by modifications to the program. A critical section of a program can be associated with each performance attribute such that if this critical section is modified, the corresponding performance attribute may be affected. For example, if the performance attribute under consideration is the execution time between when a module begins execution and when it transmits a message, the corresponding critical section is that section of code between module invocation and transmission of the message. It should be noted that a critical section for a particular performance attribute may be part of another critical section for a different performance attribute. In this case, a modification to a critical section within another critical section can affect the corresponding performance attributes of both critical sections. The relationship of performance attributes, critical software sections, and the mechanisms for the propagation of performance changes is illustrated in Figure 12, where the directed line labeled with a mechanism connecting two performance attributes indicates a performance dependency relationship exists between the performance attributes. A directed line also connects each critical section (C.S.) with its corresponding performance attribute. It is apparent from Figure 12 that a modification to C.S.1 being also a modification to C.S.2 implies both P.A.2 and P.A.3 of Module A may be affected. Also, a change in P.A.2 of Module A may affect P.A.2 of Module B via mechanism one. Thus, the modification in Module A can affect the performance of Module B.

The identification of the critical sections corresponding to the performance attributes requires algorithms whose input includes an identifica-

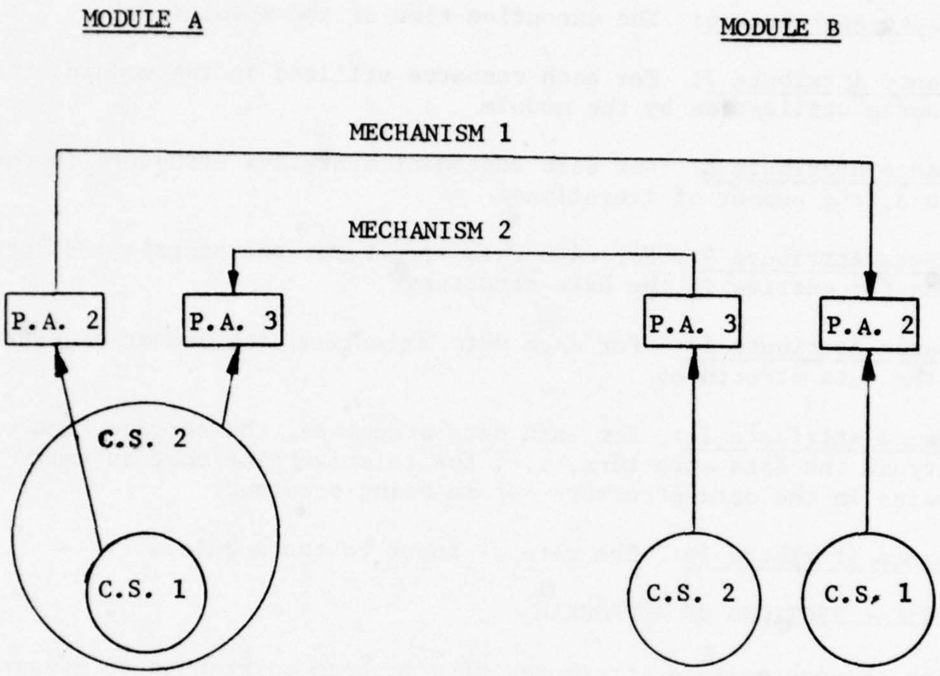


Figure 12. Relationship of performance attributes (P.A.), critical sections (C.S.) and the mechanisms for the propagation of performance changes in a program.

tion of the mechanisms in existence in the program. These algorithms for identifying the critical sections are formally described in our second report.

9. RIPPLE EFFECT OF PERFORMANCE CHANGE

The relationship of performance attributes, critical sections, and the mechanisms for the propagation of performance changes in a program forms the basis for the concept of a performance change ripple-effect as a consequence of software modification. When a critical section is modified, it may affect the corresponding performance attributes. A change in these performance attributes may then ripple, i.e. affect other performance attributes via any applicable mechanisms.

A performance dependency relationship is defined to exist between two performance attributes if a change in one of the performance attributes may affect the other performance attribute via one of the mechanisms for the propagation of performance changes. Thus, the determination of the performance dependency relationships requires an identification of the mechanisms in existence in the program. In our second report, rules for formulating these performance dependency relationships are presented in detail. The format of these rules consists of the identification of a performance dependency relationship between a performance attribute of one type of a module and a particular performance attribute of another module with the restriction that the modules are involved in a performance dependency relationship via one of the mechanisms. For example, one rule states that a performance dependency relationship exists between performance attribute 2 of module X and performance attribute 2 of module Y if modules X and Y are in a performance dependency relationship via the shared resources mechanism. The performance dependency relationships between performance attributes are then identified by applying these rules to the actual program being analyzed. The performance dependency relationships are then saved for utilization during the maintenance phase of the program.

10. MAINTENANCE TECHNIQUE FOR PREDICTING WHICH PERFORMANCE REQUIREMENTS ARE AFFECTED BY THE MAINTENANCE ACTIVITY

The maintenance process can be improved if maintenance personnel are supplied with information enabling them to incorporate performance considerations in their criteria for selecting the type and location of program modifications to be made. This information is provided by the development of a maintenance technique for predicting which performance requirements in the program may be affected by a proposed modification. The prediction of which performance requirements may be affected by a program modification is a difficult task due to the size and complexity of design of many large-scale software systems. Thus, the significance of this maintenance technique lies in its ability to trace repercussions introduced by maintenance changes and predict which performance requirements may be affected by the change. The technique developed here is applicable to all types of large-scale software systems possessing performance requirements including multiprocessing systems. In the next section, we are going to present a general framework for such a technique. The formal description of the algorithms composing this technique as well as the proofs of the correctness of these algorithms is included in our second report. Also included in that report is a demonstration of the maintenance technique during the maintenance phase of a typical program.

10.1 The General Framework of the Maintenance Technique

The maintenance technique consists of two phases. The first phase analyzes the program and produces a data base which is saved for utilization in the second phase of the technique when maintenance activity is in progress. Thus, the first phase can be performed as soon as the program has successfully passed its acceptance tests and is entering the operational stage of its life cycle. This first phase of the maintenance technique consists of the following steps:

PHASE ONE

Step 1: The performance requirements for the program must be decomposed into the key performance attributes which contribute to the preservation or violation of the performance requirements. The decomposition of a performance requirement quantitatively into the effect of its corresponding performance attributes is a very complex task which is not attempted in this technique. Instead, the decomposition is qualitative in nature, i.e. performance attributes are identified which contribute to the preservation or violation of performance requirements without consideration of their relative magnitude towards the performance requirements. This simplification is justified because this maintenance technique attempts to identify performance requirements which may be violated due to the maintenance effort, and does not attempt to analytically confirm whether or not a performance requirement is actually violated. The process of identifying key performance attributes contributing to a performance requirement is not a difficult process and can be accomplished manually. As requirement statement languages continue to develop, it is likely that this process can be automated. Figure 13 is an expansion of Figure 12 which includes a description of the relationship of performance requirements, performance attributes, critical sections, and the mechanisms for the propagation of performance changes in a program. In this figure, the directed line labeled with a mechanism connecting two performance attributes indicates a performance dependency relationship exists between the performance attributes. A directed line also connects each critical section with its corresponding performance attributes. A dotted line is used to connect each performance attribute with a performance requirement which may be affected if the performance attribute is changed.

Step 2: Determine which mechanisms for the propagation of performance changes are present in the program.

Step 3: Identify critical sections corresponding to the performance attributes identified in Step 1.

Step 4: Identify performance dependency relationships between performance

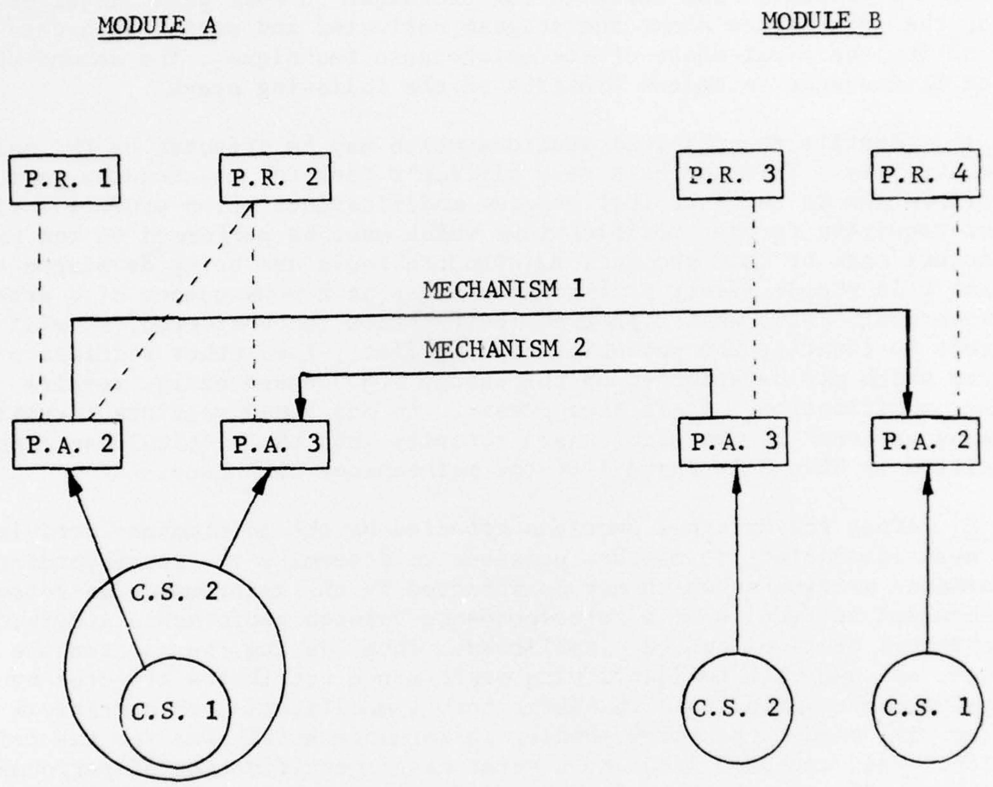


Figure 13. Relationship of performance requirements (P.R.), performance attributes (P.A.), critical sections (C.S.), and the mechanisms for the propagation of performance changes in a program.

attributes present in the program.

PHASE TWO

The second phase of this maintenance technique is applied during the maintenance process. The input to the technique in this phase requires all of the information about the program collected and stored in a data base during the first phase of the maintenance technique. The second phase of the maintenance technique consists of the following steps:

Step 1: Identify the critical sections which may be affected by the maintenance activity. This can be a very difficult task for maintenance personnel to perform due to the fact that program modifications often produce a ripple-effect requiring further modifications which must be performed on the program. In another task of this project, maintenance tools are being developed to predict this ripple effect of logical changes as a consequence of a program modification. Thus, when a program modification is considered, it will be possible to identify the potential ripple-effect, i.e. other sections of the program which may be affected by the change and, consequently, require further modification. It is then possible to map these sections identified as being affected by the maintenance activity into the critical sections identified in Step 3 in Phase 1 of the maintenance technique.

Step 2: After the critical sections affected by the maintenance activity have been identified, it is then possible to determine the corresponding performance attributes which may be affected by the maintenance activity. As discussed in Section 8, a correspondence between performance attributes and critical sections can be established. Thus, during the maintenance process, one approach to identifying performance attributes affected by the maintenance activity is to assume that a modification to a critical section will affect the corresponding performance attributes for the critical section. This approach leads to a worst-case identification of performance attributes affected by the software modification. A second approach to identifying performance attributes affected by the software modification is to solicit the help of the maintenance personnel implementing the modification. After the critical sections affected by the proposed modification have been identified, the corresponding performance attributes for these critical sections can then be displayed to the maintenance personnel. The maintenance personnel can then decide whether or not the displayed performance attribute will be affected based upon his knowledge of the proposed maintenance modification.

Step 3: Utilizing the performance dependency relationships established in Step 4 in Phase 1, identify all the performance attributes affected by changes to the performance attributes identified in Step 2 of this phase. The objective of this step is to predict all the performance attributes throughout the system which are affected by the program modification. This prediction is basically a worst-case prediction except for the refinements introduced in the last step.

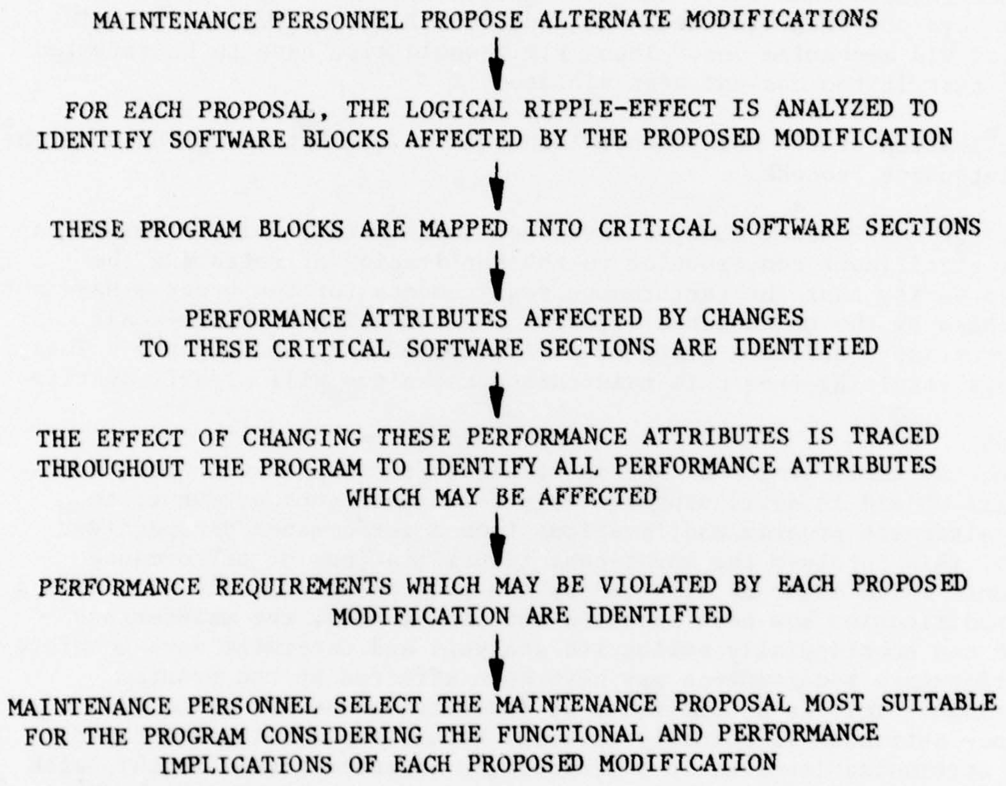
Step 4: Utilizing the list of performance attributes identified in the last step, identify those performance requirements which may be affected by the maintenance activity and, thus, should be retested to insure that they have not been violated. The performance requirements can be easily identified by the traceability of the decomposition of the performance requirements into the performance attributes in Step 1 of Phase 1.

The important steps of this maintenance technique are summarized and put into perspective within the maintenance process in Figure 14. From the example shown in Figure 13, the maintenance technique could be used to predict the performance implications of modifying C.S.1 of Module A. In this example, P.A.2 and P.A.3 of Module A would be affected. Thus, performance requirements P.R.1 and P.R.2 would have to be retested to insure that they have not been violated. In addition. P.A.2 of Module B would be affected via mechanism one. Thus, P.R.4 would also have to be retested to insure that it too has not been violated.

10.2 Application of the Maintenance Technique to the Retesting Phase of the Maintenance Process

After the maintenance changes have been implemented, this technique can provide a significant contribution to the application of retesting the program to verify that the performance requirements for the program have not been violated by the maintenance effort. The retesting of large-scale complex programs requires a great deal of time, effort, and expense. Thus, any savings resulting from this maintenance technique will clearly justify its use.

During the early stages of the maintenance process, this technique was utilized as an aid in developing criteria for maintenance personnel to evaluate alternate program modifications from a performance perspective. Basically, this involved the worst-case identifications of performance requirements which might be affected by the program modifications. After a program modification has been selected and implemented, the maintenance technique can substantially refine its analysis and determine more accurately which performance requirements may have been affected by the program modifications. This is accomplished by determining whether or not a performance attribute is actually affected before implicating other performance attributes involved in a performance dependency relationship with the given attribute. In other words, if a dependency relationship exists between performance attributes one and two, performance attribute two need not be examined for changes if it has been determined that performance attribute one is not affected by the maintenance activity. Thus, the preliminary results of some of the early retesting efforts may be decisive in the determination of the scale of retesting which remains to be done. This technique is summarized and put into perspective within the maintenance process in Figure 15. It should be noted that if a violation of a performance requirement occurs, it requires further software maintenance in order to satisfy the performance requirement, and the entire process must be repeated.



MAINTENANCE PERSONNEL PROPOSE ALTERNATE MODIFICATIONS

FOR EACH PROPOSAL, THE LOGICAL RIPPLE-EFFECT IS ANALYZED TO IDENTIFY SOFTWARE BLOCKS AFFECTED BY THE PROPOSED MODIFICATION

THESE PROGRAM BLOCKS ARE MAPPED INTO CRITICAL SOFTWARE SECTIONS

PERFORMANCE ATTRIBUTES AFFECTED BY CHANGES TO THESE CRITICAL SOFTWARE SECTIONS ARE IDENTIFIED

THE EFFECT OF CHANGING THESE PERFORMANCE ATTRIBUTES IS TRACED THROUGHOUT THE PROGRAM TO IDENTIFY ALL PERFORMANCE ATTRIBUTES WHICH MAY BE AFFECTED

PERFORMANCE REQUIREMENTS WHICH MAY BE VIOLATED BY EACH PROPOSED MODIFICATION ARE IDENTIFIED

MAINTENANCE PERSONNEL SELECT THE MAINTENANCE PROPOSAL MOST SUITABLE FOR THE PROGRAM CONSIDERING THE FUNCTIONAL AND PERFORMANCE IMPLICATIONS OF EACH PROPOSED MODIFICATION

Figure 14. The framework of a maintenance technique in predicting which performance requirements are affected by the maintenance activity

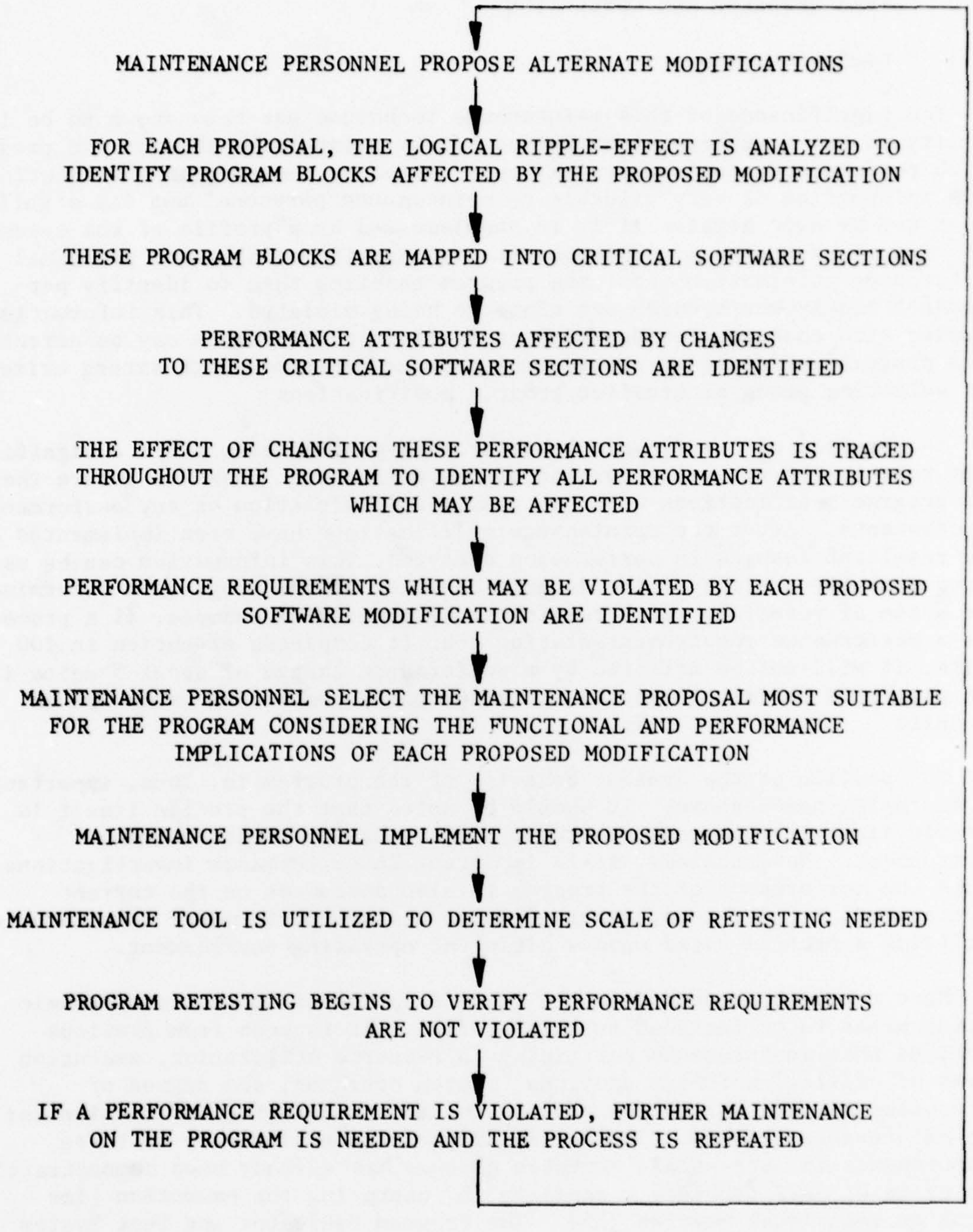

MAINTENANCE PERSONNEL PROPOSE ALTERNATE MODIFICATIONS
FOR EACH PROPOSAL, THE LOGICAL RIPPLE-EFFECT IS ANALYZED TO IDENTIFY PROGRAM BLOCKS AFFECTED BY THE PROPOSED MODIFICATION
THESE PROGRAM BLOCKS ARE MAPPED INTO CRITICAL SOFTWARE SECTIONS
PERFORMANCE ATTRIBUTES AFFECTED BY CHANGES TO THESE CRITICAL SOFTWARE SECTIONS ARE IDENTIFIED
THE EFFECT OF CHANGING THESE PERFORMANCE ATTRIBUTES IS TRACED THROUGHOUT THE PROGRAM TO IDENTIFY ALL PERFORMANCE ATTRIBUTES WHICH MAY BE AFFECTED
PERFORMANCE REQUIREMENTS WHICH MAY BE VIOLATED BY EACH PROPOSED SOFTWARE MODIFICATION ARE IDENTIFIED
MAINTENANCE PERSONNEL SELECT THE MAINTENANCE PROPOSAL MOST SUITABLE FOR THE PROGRAM CONSIDERING THE FUNCTIONAL AND PERFORMANCE IMPLICATIONS OF EACH PROPOSED MODIFICATION
MAINTENANCE PERSONNEL IMPLEMENT THE PROPOSED MODIFICATION
MAINTENANCE TOOL IS UTILIZED TO DETERMINE SCALE OF RETESTING NEEDED
PROGRAM RETESTING BEGINS TO VERIFY PERFORMANCE REQUIREMENTS ARE NOT VIOLATED
IF A PERFORMANCE REQUIREMENT IS VIOLATED, FURTHER MAINTENANCE ON THE PROGRAM IS NEEDED AND THE PROCESS IS REPEATED

Figure 15. Application of the maintenance technique in the prediction of performance requirements affected by the maintenance activity and in the retesting phase.

11.0 FUTURE RESEARCH AND CONCLUSION

11.1 Dynamic Analysis

The significance of this maintenance technique has been shown to be its ability to trace repercussions introduced by maintenance changes and predict which performance requirements may be affected by the program modifications. This information is very valuable to maintenance personnel but its significance can be even greater if it is supplemented by a profile of the dynamic behavior of the program. This profile can provide maintenance personnel with performance information about the program enabling them to identify performance requirements which are close to being violated. This information coupled with that predicting which performance requirements may be affected by a program modification provide maintenance personnel with strong criteria for selecting among alternative program modifications.

The profile of the dynamic behavior of a program also plays a significant role in the retesting portion of the maintenance phase to insure that the program modifications have not resulted in violation of any performance requirements. After the maintenance modifications have been implemented and the resultant changes in performance analyzed, this information can be used along with the profile of the dynamic behavior of the program to determine the scale of retesting which remains to be done. For example, if a process has a performance requirement stating that it completes execution in 100 units, it will not be affected by a performance change of about 5 units if the program's profile indicates it is currently completing execution in 75 units.

The profile of the dynamic behavior of the program is, thus, important in the maintenance phase. It should be noted that the profile itself is dynamic since it only reflects the dynamic behavior in the current environment. Nevertheless, it is important in performance investigations since the performance of the program is also dependent on the current environment. It is, thus, meaningless to analyze performance considerations utilizing a profile based upon a different operating environment.

More research is needed in the identification of appropriate dynamic measurements to be included in this profile. It is seen from previous sections that measurements pertaining to resource utilization, execution times of critical software sections, system overhead, and degree of saturation of data structures provide the most meaningful information for the maintenance process. The feasibility of collecting many of these measurements in large-scale software systems has already been demonstrated. For example, JAVS provides a facility for capturing the execution time spent in individual modules [25]. The Program Evaluator and Test System developed by Stucki [26] also provides relative timing on the subroutine level. System overhead has also been studied for some time and both hardware and software measuring techniques exist to identify many of its sources. Measurements pertaining to resource utilization by software processes have also been recorded using software probes. Data pertaining to the time when a resource is requested and when that request is actually satisfied have been collected on large-scale software systems with a degradation of system performance due to resources committed to probe operation not exceeding 5%. For example Figure 16 illustrates the types of measurements that can be gathered for an executing process. In the figure, execution times between requests as well as probabilities that particular branches from decision nodes are executed appear in the graph [27]. These types of measurements would be important in formulating the profile of the dynamic behavior of the program most applicable to the maintenance process.

11.2 Figure-of-Merit for Program Maintainability from a Performance Perspective

The theoretical foundation for this maintenance technique also forms the basis for the development of a figure-of-merit for the program maintainability of a system from a performance perspective. This figure-of-merit is a measure of the impact of maintenance activity on the performance requirements of the program. This figure-of-merit could be computed for the program as early as the design stage. It could then be refined with information available after implementation to provide a more precise measure of the maintainability from a performance perspective. It is seen from the previous sections that several factors might contribute to this figure-of-merit. The first involves the degree of performance dependency relationships in existence as indicated by the mechanisms for the propagation of performance changes. The second involves the extent and stringency of the performance requirements imposed upon the program. The performance of a program is a subject which ranges from quantitative analyses to qualitative judgments [28]. Thus, the figure-of-merit must be based from a user perspective. The third involves the profile of the dynamic behavior of the program. The current behavior of the program in perspective to its performance requirements can provide insight into the degree of performance changes that can be tolerated without a performance requirement violation. For example, if the program is operating near a point of saturation, any performance changes could lead to performance requirement violations. More research is needed in this area to identify additional factors contributing to the figure-of-merit and to integrate these factors into a meaningful figure.

Most of the factors utilized in the computation of the figure-of-merit would be applicable to the computation of a figure for the complexity of modification to the program. More research is needed in the identification of additional factors contributing to this figure. The complexity measure would be valuable to maintenance personnel by providing a quantitative comparison of potential program modifications in terms of their maintenance characteristics from a performance point of view.

11.3 Application to Design Phase

Throughout this report, the major emphasis has been upon the propagation of performance changes as a consequence of program modifications. The results of this investigation will be valuable in evaluating current software design techniques in terms of their ability to produce good maintenance characteristics from a performance point of view.

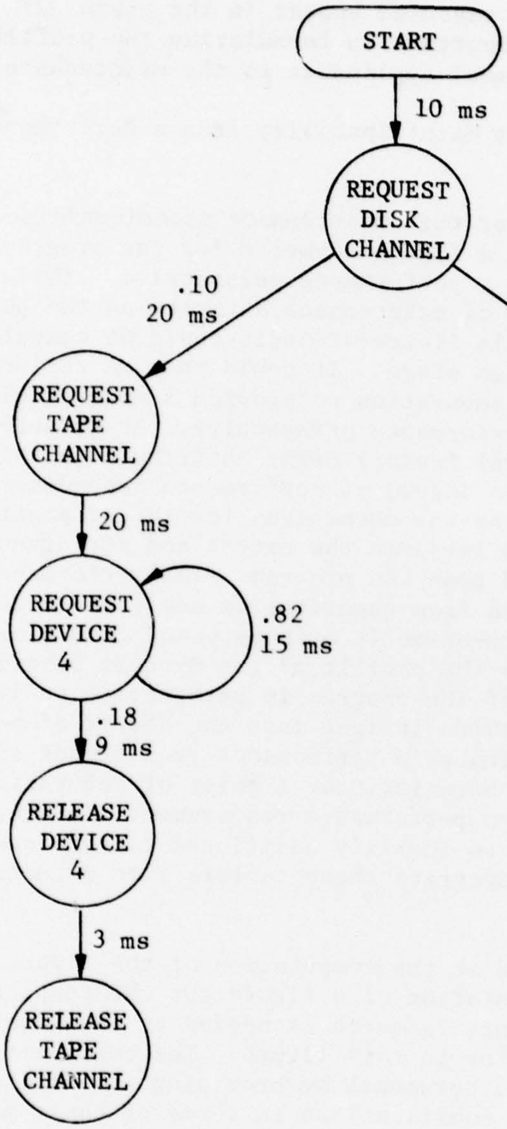


Figure 16. The directed graph of resource utilization of a process.

11.4 Conclusions

The process of developing complex large scale software systems possessing performance requirements is costly, excessively time-consuming, and difficult to manage. This process frequently leads to systems which are unreliable, non-responsive to user-requirements, and logically too obscure to be readily analyzed or maintained [29]. Yet these software systems must be maintained, and the magnitude of this maintenance in terms of the total software effort is very large. Thus, maintenance techniques are needed that are specifically designed to predict the effect of software modifications and indicate test cases required for program retesting [30]. The maintenance technique outlined in this report is designed with these objectives in mind and should significantly aid maintenance personnel in maintaining software systems.

REFERENCES

[1] Mills, H. D., "Software Development," IEEE Trans. on Software Engineering, Vol. SE-2, No. 4, December 1976, pp. 265-273.

[2] Rye, P., Bamberger, F., Ostanek, W., Brodeur, N., and Goode, J., Software Systems Development: A CSDL Project History, RADC-TR-77-213, pp. 33-41. (A042186)

[3] Goodenough, J. B., et al., "The Effect of Software Structure on Software Reliability, Modifiability, and Reusability: A Case Study and Analysis," NTIS AD 787 307, July 1974, p. 82.

[4] McCall, J. A., Richards, P. K., and Walters, G. F., Factors in Software Quality, Volumes I, II, and III, General Electric Company, pp. 2-3, 3-5, 7-9.

[5] Yourdon, E. and Constantine, L., Structured Design, Yourdon, Inc., 1976, p. 392.

[6] Belford, P. C., Donahoe, J. D., and Heard, W. J., "An Evaluation of the Effectiveness of Software Engineering Techniques," Digest of Papers, COMPCON '77 (Fall), pp. 259-269.

[7] Herd, J. H., Postak, J. N., Russell, W. E., and Stewart, K. R., Software Cost Estimation Study, Volume I, RADC-TR-77-220, June 1977, pp. 88-89. (A042264)

[8] Doty, D. L., Nelson, P. J., and Steward, K. R., Software Cost Estimation Study, Volume II, RADC-TR-77-220, August 1977, p. A-5. (A044609)

[9] Parnas, D. L., "A Technique for the Specification of Software Modules with Examples," Comm. of ACM, Vol. 15, May 1972, pp. 330-336.

[10] Liskov, B. H. and Zilles, S. N., "Specification Techniques for Data Abstractions," IEEE Trans. on Software Engineering, Vol. 1, No. 1, March 1975, pp. 7-19.

[11] White, J. R. and Booth, T. L., "Towards an Engineering Approach to Software Design," Proceedings of the Second International Conference on Software Engineering, 1976, pp. 214-222.

[12] Gilkey, T. J., White, J. R., and Booth, T. L., "Performance Analysis as a Practical Software Design Tool," Proceedings of COMPSAC '77, pp. 428-435.

[13] Schneidewind, N. F., "Modularity Considerations in Real-Time Operating System Structures," Naval Postgraduate School, pp. 7-11.

[14] Branscomb, L. M., "The Everest of Software," Proceedings of the Symposium on Computer Software Engineering, 1976, pp. xvii-xx.

[15] Sholl, H. A. and Booth, T. L., "Software Performance Modeling Using Computation Structures," IEEE Trans. on Software Engineering, Vol. SE-1, No. 4, December 1975, pp. 414-420.

[16] Storey, T. and Todd, S., "Performance Analysis of Large Systems," Software Practice and Experience, Vol. 7, 1977, pp. 363-369.

[17] Davis, C. G. and Vick, C. R., "The Software Development System," IEEE Trans. on Software Engineering, Vol. SE-3, No. 1, January 1977, pp. 69-84.

[18] Bell, T. E. and Bixler, D. C., "A Flow-Oriented Requirements Statement Language," Symposium on Computer Software Engineering, 1976, pp. 109-123.

[19] Alford, M. W. and Burns, I. F., "R-Nets: A Graph Model for Real-Time Software Requirements," Symposium on Computer Software Engineering, 1976, pp. 97-107.

[20] Belford, P. C., "Specifications: A Key to Effective Software Development," Proceedings of the Second International Conference on Software Engineering, 1976, pp. 71-79.

[21] Balkovich, E., "Research Towards a Technology to Support the Specifications of DPSPR," Proceedings of the Second International Conference on Software Engineering, 1976, pp. 110-115.

[22] Salter, K. G., "A Methodology for Decomposing System Requirements into Data Processing Requirements," Proceedings of the Second International Conference on Software Engineering, 1976, pp. 91-102.

[23] Kosy, D. W., "Air Force Command and Control Information Processing in the 1980's: Trends in Software Technology," June 1974, p. 18.

[24] Valette, R. and Diaz, M., "Top-Down Formal Specification and Verification of Parallel Control Systems."

[25] JAVS Technical Report Reference Manual, RADC-TR-77-126, Vol. 2, April 1977. (A040104)

[26] Stucki, L. G., "Automatic Generation of Self-Metric Software," Proceedings of 1973 IEEE Symposium on Computer Software Reliability, pp. 94-101.

[27] Anderson, J. W. and Browne, J. C., "Graph Models of Computer Systems: Application to Performance Evaluation of an Operating System," Proceedings of the International Symposium on Computer Performance Modeling, Measurement, and Evaluation, 1976, pp. 187-199.

[28] Kuck, D. J. and Kumar, B., "A System Model ıor Computer Performance Evaluation," Proceedings of the International Symposium on Computer Performance Modeling, Measurement, and Evaluation, 1976, pp. 187-199.

[29] DeWolf, B. J., A Methodology for Requirements Specification and Preliminary Design of Real-Time Systems, The Charles Stark Draper Laboratory, Inc., July 1977, pp. 1-1 to 2-20.

[30] Ramamoorthy, C. V., and Ho, S. F., "Testing Large Software with Automated Software Evaluation Systems," Current Trends in Programming Methodology Volume II, Prentice-Hall, Inc., edited by Raymond Yeh, 1977, pp. 112-150.